Extending Object Database Interfaces with Fuzziness through Aspect–Oriented Design

Miguel-Ángel Sicilia & Elena García-Barriocanal
Computer Science Dept., University of Alcalá {msicilia,elena.garciab}@uah.es

Abstract

Fuzzy logic has been used yet for extending database models to deal with vagueness in the definitions of linguistic concepts as "tall" or "long". However, the extension of existing programming interfaces for fuzziness requires a proper modularization of the underlying concerns of numerical imprecision handling. Such modularization should not interfere with existing programming practices, and they should not obscure the original design. Aspect–oriented design (AOD) enables such form of non–intrusive extensions to be added to existing software libraries. In this paper, the main design and implementation issues of such AOD–based extensions on `OJB` database libraries are briefly sketched.

1 Introduction

Fuzzy set theory provides numerical models for handling vagueness as expressed for example in sentences like "x is *tall*". In such sentences, the determination of which individuals belong to the class or set *tall* is a matter of (numerical) degree that depends on their height, as modeled in what is commonly known as "membership functions". In addition, other related mathematical frameworks have provided numerical models for other kinds of imprecision or uncertainty [13]. The integration of such models into database structures has been subject of a significant amount of research in recent years. Fuzzy databases and fuzzy querying research has resulted in diverse extensions to the relational and object data models [2, 3, 5]. Further, several implementations of fuzzy queries on top of commercial database systems or standard interfaces have been developed, e.g. [9, 17, 18].

In some modern database approaches, interfaces for *orthogonal persistence* of objects [1] are provided to the programmers, i.e. the provision of persistence is the same for all data irrespective of their type. This entails that no specific programming is required to make a type persistent. The extension for fuzziness of such kind of interfaces can be accomplished by adding elements to the query syntax and also by augmenting programming interfaces to deal with the desired fuzzy capabilities — e.g. as in [9]. In any case, several reasons point to some implementation characteristics that are required for a seamless integration with existing interfaces. On the one hand, extensions should be *strictly additive*, i.e. they should not interfere with the non–fuzzy capabilities of the programming and querying interfaces, both for the sake of backward–compatibility and of ease of learning [12]. And on the other hand, the extensions should be properly *modularized*, not obscuring the original design and architecture of the extended database libraries. The first requirement can be met through a careful design, using polymorphism and proxy objects as described in [9], and using reflective capabilities when required. Nonetheless, the second requirement calls for specialized design and implementation capabilities that allow the extension of software with cross–cutting concerns (as fuzziness can be considered with regards to data representation). Aspect–oriented design (AOD) [16] provides the required modularization capabilities for the latter issue. In this paper, the implementation issues of fuzzy extensions to object–database interfaces are approached from the perspective of AOD. Such novel approach is then put into practice through a concrete case study using `aspectj` to extend the Java–based interfaces of the `ObjectRelationalBridge (OJB)`[1] open–source libraries. Related work includes studies of fuzziness related to prototype theory of categorization [15] to classify types of software units, but no previous research exists on the introduction of fuzziness in software units through aspect–oriented techniques.

The rest of this paper is structured as follows. Section 2 describes the rationale behind using AOD to implement fuzziness as a concern in existing programming interfaces. In Section 3, the general issues of extending `OJB` interfaces with fuzziness are described. Then, a simple case of fuzzy extension is sketched in

[1]http://db.apache.org/ojb/

Section 4.

2 Fuzziness as a Concern in Object-Oriented Software

Imperfection in information should be addressed early in the lifecycle due to the specifics of uncertainty and imprecision in conceptual modeling [4], and its impact on architectural and implementation decisions — most notably in persistence and querying [9]. Information imperfection is a logical "matter of interest", according to COSMOS [14] terminology. It can be organized in several *classifications* [14], one for each of the considered principal aspects of imperfection, e.g. `Imprecision-Related`, `Uncertainty-Related`, `Inconsistency-Related` or `Hybrid`. Further subdivisions inside those categories may refer to more specific types of imperfection according to a given list like Smet's taxonomy [13]. For example, `FuzzyElement` refers to "imprecision without error" without decidability as in "age is close to 30", while `PossibleElement` refers to "happen–ability" as a kind of uncertainty. According to the kind of element in the conceptual model that is subject to imperfection, we can have additional classifications in another dimension, namely `Imperfect-Element`, and `ImperfectRelationship`, the former containing subdivisions as `Imperfect-Class`, `Imperfect-Attribute`, `Imperfect-Function` and `Imperfect-Result`, which roughly correspond to classes, attributes, methods and method results in object–oriented conceptual models. Imperfect conceptual model elements can be expressed in the domain model through extensions to the *Unified Modeling Language* (UML) notation like the one sketched in [8].

AOD can be used as the candidate detailed design technique whenever concerns cross–cut software modules, and it becomes the required option if existing libraries are required to be extended for fuzziness without changing the syntax and semantics of their interfaces. This is because *aspects* like those of `aspectj`[6] allow the proper modularization and encapsulation of concerns, avoiding tangling existing source code. Moreover, AOD enhances maintainability, since the added concerns for fuzziness are separated from "crisp" functionality, so that defects can be easily located. In addition, AOD for the implementation of fuzziness eases the production of "crisp" and "fuzzy" versions of the same software, since the features of *advice* and *introduction* contained in aspects are dynamic, and thus they can be "disabled" by simply excluding the aspects from the concrete build of the system.

3 Extending `OJB` with `aspectj`

As mentioned above, the techniques of aspect-oriented design (AOD) provide improved modularity to software systems by focusing on separation of concerns [16]. Fuzziness can be considered a cross–cutting concern for existing database processing libraries, so that it can be added to existing crisp software without altering the original programming interfaces. As a proof of concept for introducing fuzziness, the `OJB` framework has been extended by using `AspectJ`. The `AspectJ` framework [6] is an AOD extension to the Java programming language based on the concept of dynamic *pointcut* (the intersection of a number of well–defined execution points) and *advice* (code attached to specific pointcuts). Using aspects in `OJB` requires a previous recompilation of its source code version. OJB supports multiple persistence application programming interfaces (APIs), including JDO and ODMG compliant ones. However, all of them are built on top of a persistence kernel, so that it makes sense to concentrate first on it, and later address higher–level interfaces. In addition, a principle of "minimum difference" with existing interfaces and programming idioms is followed, in an attempt to maximize the *usability* of the extensions in the sense described in [12].

3.1 Extending metadata handling

Metadata handling in `OJB` is centralized in the `MetadataManager` class, implementing a *singleton* pattern that can be used to obtain a reference to the `DescriptorRepository` instance containing object mapping and manipulation information for persistent objects. The class needs to be extended to deal with the required metadata describing fuzzy constructs. The better way to do it is by merging standard metadata descriptions with fuzzy ones. This can be accomplished by invoking the `mergeDescriptorRepository` method after a (successful) call to the `MetadataManager.init` method at construction time, by capturing its *pointcut* through an around *advice*. The following code fragment sketches an aspect encapsulating such processing:

```
public aspect FuzzyMetadataManagement {
  DescriptorRepository drp;
  void around (MetadataManager m):
      target(m) && call(* MetadataManager.init(..)){
        try{ proceed(m);
        }catch(MetadataException e){throw e;}
        drp = loadFuzzyDescRepository();
        m.mergeDescRepository(drp);
    }
    private DescriptorRepository
            loadFuzzyDescRepository(){
        DescriptorRepository dr =
                    new DescriptorRepository();
        //  load descriptor repository..
        return dr;
    }    //...}
```

The `loadFuzzyDescRepository` method simply carries out the processing of the fuzzy schema description residing in a XML file, similar to that described in [9]. It should be noted that neither inheritance nor reflection would have been enough to make this change without modifying existing libraries, since at least the `getInstance` method of `MetadataManager` would had to be changed to instantiate a new subclass or build it dynamically.

3.2 Describing `Imperfect-`Elements

The current interface of `DescriptorRepository` makes use of `ClassDescriptor` for the object–relational mapping descriptions, which in turn uses `FieldDescriptor` to specify the storage of class attributes. Both elements can be extended to their fuzzy counterparts through subclassing, and other conceptual data elements like associations [10] can be derived from the common super–class `DescriptorBase` which is an open–ended hook for them. `Imperfect-Class` and `Imperfect-Attribute` concerns can be implemented by extending the `ClassDescriptor` and `FieldDescriptor` classes respectively with methods to query for the data mapping of the membership degrees, required to implement querying and storage functionality. Figure 1 depicts some of the details of such extension as a `UML` diagram. As showed in Figure 1, the `AttributeDescriptorBase` is used as an intermediate extension point for persistence mappings that are not necessarily relational, while `FieldDescriptor` provides the specific details of the relational mapping. `FuzzyFieldDescriptor` is provided as the base class for any relational–mapping of imperfect attributes, and `FuzzyNumericFieldDescriptor` is one of its subclasses representing the mapping for (triangular) fuzzy numbers. `FuzzyClassDescriptor` encapsulates the details for the relational mapping of fuzzy classes and sub–classes, with the possible variants described in [9]. *Extensional* mappings store

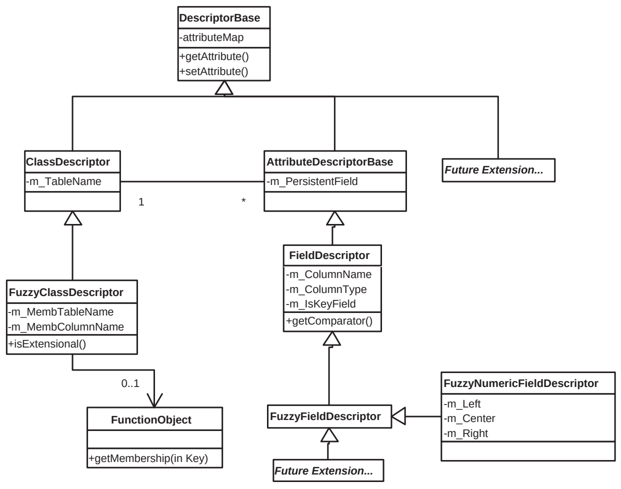

Figure 1: Extension of `OJB` description classes

explicitly the membership degrees for each object in the database, while *intensional* mappings provide a `FunctionObject` instance encapsulating arbitrarily complex computations of membership degrees for the class. In this latter case, Java's reflection capabilities enable the specification of a `FuzzyObject` subclass in the configuration file that is instantiated dynamically at run–time.

3.3 Basic fuzzy storage

Object storage both in `ODMG` and `JDO` mappings proceeds in cascade, storing the graph of references starting from the object being stored. For example, JDO provides a method `makePersistent` in the `PersistentManager` interface to make concrete instances persistent, and it also provides persistence by "reachability", so that any instance linked to a persistent one (transitively) is also made persistent. The underlying core `OJB` API ultimately uses the `store` methods in the `PersistenceBroker` interface to resolve those calls, which explicitly handles also the storage of `Collection` implementing classes. The storage of fuzzy classes only requires code modifications for *extensional* fuzzy classes, in which membership degrees are explicitly stored. To do so, variants of the `store` methods with an additional parameter are required. `Aspectj` *introductions* can be used to accomplish such extension, avoiding subclassing such a complex class like `PersistenceBrokerImpl`. The aspect can be targeted to the interface `PersistenceBroker` to guarantee that future implementation classes also are provided with fuzzy storage methods. The following code fragment sketches such extension:

```
public aspect FuzzyStorageHandler {
```

```
public void PersistenceBroker.store(
   Object obj, Double m,String fuzzyClass)
            throws PersistenceBrokerException;
public void PersistenceBrokerImpl.store(
    Object obj, Double m,String fuzzyClass)
    throws PersistenceBrokerException {
    store(obj); storeMembership(obj,m,fuzzyClass);
}
public void
    PersistenceBrokerImpl.storeMembership(
         Object obj, Double m, String fc)
         throws PersistenceBrokerException {
 FuzzyClassDescriptor c = (FuzzyClassDescriptor)
  descriptorRepository.getDescriptorFor(fc);
 if (c.isExtensional()){
   // call access layer to store m...
}  } }
```

The `fuzzyClass` attribute is required since a single object may belong to more than one fuzzy class with different degrees, i.e. multiple classification outside of the capabilities of the programming language is assumed. The storage of fuzzy attributes is delegated in `PersistenceBrokerImpl` to a `JdbcAccess` interface which acts as a layer encapsulating the construction of SQL sentences from metadata descriptors and actual objects to be stored. The delegation chain can be followed through the `StatementManager` — responsible for the value `binding` process — class to the `StatementsForClass` interface and implementation – which only provides caching for statement templates — and to the `SqlGenerator` interface and implementation, which actually build the SQL queries from class descriptions by delegating to a number of classes, one for each type of SQL sentence. In consequence, it is in the `SqlUpdateStatement` where the actual sentence creation logic resides, where it can be seen that no modification is required to store fuzzy classes, since it relies in the (extended through polymorphism) behavior of `ClassDescription`.

In contrast, the storage of fuzzy fields as those described by `FuzzyNumericFieldDescriptor` require dynamic extension of the methods `appendListOfColumns` and `appendListOfValues` since they assume a single table column for each field. This can be accomplished by an aspect design as the one sketched in what follows:

```
public aspect FuzzySqlFieldHandling{
  List around (SqlInsertStatement i):
    target(i) && args(cld) && args(buf)
    && call(List SqlInsertStatement.
       appendListOfColumns(
       ClassDescriptor cld, StringBuffer buf)){
     List aux=null;
     try{  aux =  proceed(m, cld, buf);
     }catch(Exception e){throw e;}
     if (cld.getClass().equals( new
        FuzzyNumericFieldDesc().getClass())){
        // add columns to SQL INSERT in buf...
     }   }  // ...}
```

The *around* advice is able to dynamically extend the behavior of the SQL–forming methods by manipulating parameters and return values.

3.4 Fuzzy querying through aspects

The abstract `SqlQueryStatement` class and its concrete subclass `SqlSelectStatement` together implement the generation of SQL SELECT clauses from queries. Queries for fuzzy classes and fuzzy attributes can be formed by extending `getStatement` invocations by using an *around* advice and an implementation technique similar to the one described above for insertions. In addition, it is required that the membership degrees of the result collections are differentiated from standard attributes of the objects. To do so, query results returning from methods like `getCollectionByQuery` in `PersistenceBroker` need to be wrapped into objects representing pairs $(o, \mu_q(o))$. This must be done at the level of the implementation of the method `getCollectionByQuery` in `PersistenceBrokerImpl` since memberships come from the access layer as conventional retrieved database columns.

```
public aspect FuzzyObjectWrapping{
    Collection around (PersistenceBroker p):
        target(p) && args(cld) && args(buf)
        && call(Collection
     PersistenceBroker.getCollectionByQuery(
          Query query)){
           Collection aux=null;
          try{
              aux =  proceed(p, query);
          }catch(PersistenceBrokerException e){throw e;}
        return this.wrapResultCollection(aux);
    }
    // ...}
```

Fuzzy query criteria instead of crisp ones can be introduced by extending the `Criteria` class and associated code. This can be accomplished alternatively by introducing additional methods to `Criteria` or by subclassing it. Fuzzy query results can be processed by casting to a class representing the pairs, resulting in a programming idiom like that described in [12]:

```
it = e.fuzzyIterator();
while (it.hasNext()){
   FuzzyObject aux = (FuzzyObject) it.next();
   X anX = (X) aux.getObject();
   double mu = aux.getMembership();
   // do processing...
 }
```

4 Example: Adding Fuzzy Classes and Fuzzy Numbers

The general design issues provided in the previous section can be used to implement a wide range of fuzzy extensions. In this section, we focus on two simple extensions for the sake of illustration, providing some important implementation details. Concretely, we will extend `OJB` with fuzzy classes and fuzzy numbers as attributes of classes, so that simple flexible queries can be issued through standard means. The domain used as a case study is that of market segmentation under fuzziness, taken from [11]. A conceptual model for the basic definitions in the case study is provided in Figure 2.

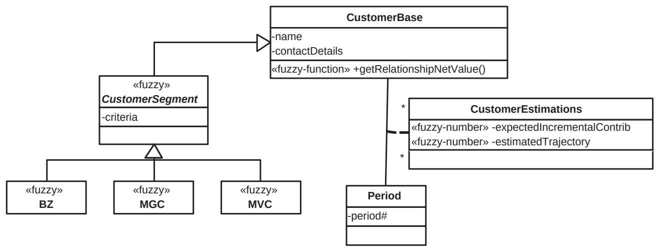

Figure 2: Main conceptual elements

In Figure 2 customers are instances of `CustomerBase`, and the estimated value of their relationship with the company is described from the marketing perspective in terms of estimations about duration (loyalty) and estimated increase in purchase volume for each period in the medium–term forecast. Both estimations are imprecise *fuzzy numbers* as marked by the stereotype `<<fuzzy-number>>`. Triangular fuzzy numbers can be represented by triples of real numbers (a, b, c) where $a < b < c$ and it is assumed that it represents a vague notion of real number roughly as "between a and c, and very close to b". The net value of their relationship is computed from those estimations by an algorithm producing also imprecise results (`<<fuzzy-function>>`), so it returns fuzzy numbers as return values. Such values is used by a process of fuzzy clustering not covered here that produce *fuzzy classes* BZ (below zero), MGC (most growable customers) and MVC (most valuable customers). Such classes are defined by overlapping membership functions as described in [11].

The system then uses the values for a sequences of periods to compute the membership degree of each customer for each of the classes (as in [11]), storing it in a explicit, extensional way, since marketing experts are able to change them due to other factors that may affect the relationship trajectory.

The storage of the example is used by specifying an XML schema like the following:

```
<class-descriptor  class="CustomerEstimations"
     table="CUSTOMER_EST" >
   <field-descriptor
   name="expectedIncContrib"
   column="EXP" left="EXP0" right="EXP1"
   type="FuzzyNum" att-left="left"
   att-center="center" att-right="right"
    primarykey="false" />
   ...
</class-descriptor>
<class-descriptor class   class="MGC"
  table="MGC" type="fuzzy" extensional="true"
  membershipTable="CUSTOMER_BASE" membField="MGC">
   ...
</class-descriptor>
```

Basically, schema definitions are `OJB`–like schemas with extended data mapping attributes and elements intended to be processed by `loadFuzzyDescRepository`. Fuzzy attributes (numbers) are mapped in the simplest way, by explicitly declaring the properties of the class that hold the left, center and right points describing the fuzzy number, so that the appropriate `getX()` methods could be invoked through reflection in the access layer.

Queries can be issued to the persistence layer by standard means provided in the core interfaces. For example, the following query returns a fuzzy subset of "BZ" (a subset of the crisp class `CustomerBase`) that has values (approximately) greater than 4.3.

```
   broker.wrapFuzzySubsets();
   Criteria criteria = new Criteria();
   criteria.addGreaterOrEqualThan(
     "expectedIncContrib", new Double(4.3));
   QueryByCriteria query = new QueryByCriteria(
           CustomerBase.class, criteria, "BZ");
   Collection res=broker.getCollectionByQuery(query);
```

It should be noted that higher–level interfaces like `Jdo` can be used alternatively to carry out fuzzy queries. For example, the following `Jdo` query returns the fuzzy subset of MVC filtered (in a crisp way) by state and area.

```
String filter =
   "contactDetails.state == state && " +
   "contactDetails.area > area";
Extent extent = pm.getExtent(CustomerBase.class,
 true, "asc;fuzzySubset=MVC");
Query query = pm.newFuzzyQuery(extent, filter);
((FuzzyQuery)query).interpretAllFuzzy();
```

```
... Collection result =
(Collection)query.execute( "Georgia", "200");
```

In the above example, the "MVC" string encoded in the parameters to `getExtent` is used to specify the class descriptor associated to the actual Java class `CustomerBase`, and the `interpretAllFuzzy()` method explicitly forces the wrapping of query results for membership processing.

5 Conclusions

Fuzziness can be considered as a separate cross–cutting concern in existing software, and in consequence, AOD techniques provide a convenient framework to implement fuzzy extensions to existing libraries. As a proof of concept for such approach, the AOD extension of the `OJB` libraries using `aspectj` has been described, along with a concrete case study regarding implementations of class and attribute concerns for fuzziness. The resulting design combines inheritance and aspects to come up with an extension that entails no modifications to existing `OJB` source code. Concretely, metadata processing is weaved at initialization and querying times, and polymorphism is used to provide alternate metadata and query result processing idioms that handle fuzziness, achieving full backwards compatibility with existing code.

References

[1] Atkinson, M. P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S.: An Orthogonally Persistent Java. *ACM Sigmod Record*, 25(4), 1996

[2] Bosc, P., Pivert, O.: Fuzzy Querying in Conventional Databases, In: Zadeh, L., Kacprzyk, J. (eds.): *Fuzzy Logic for the Management of Uncertainty.* John Wiley, New York, 1992, 645–671

[3] Buckles, B.P., Petry, F.E: A Fuzzy Representation of Data for Relational Databases. *Fuzzy Sets and Systems* 7, 1982, 213–226

[4] Chen, G.: *Fuzzy logic in data modeling : semantics, constraints, and database design.* Kluwer Academic Publishers, 1998

[5] De Caluwe, R. (ed.): *Fuzzy and Uncertain Object-Oriented Databases, Concepts and Models.* World Scientific, Singapore, 1997

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G.: An Overview of AspectJ. In: *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2001

[7] Russell, C. et al.: *Java Data Objects (JDO) Version 1.0*, proposed final draft, Java Specification Request JSR000012, 2001

[8] Sicilia, M. A., García, E., Gutiérrez, J. A.: Integrating fuzziness in object oriented modelling languages: towards a fuzzy-UML. In: *Proceedings of the International Conference on Fuzzy Sets Theory and its Applications (FSTA)*, 2002, 66-67

[9] Sicilia, M.A., García, E., Díaz, P. and Aedo, I.: Extending Relational Data Access Programming Libraries for Fuzziness: The fJDBC Framework. *Lecture Notes in Computer Science* 2522, Springer, 2002, 314–328

[10] Sicilia, M.A., Gutiérrez, J.A., García, E.: Designing Fuzzy Relations in Orthogonal Persistence Object-Oriented Database Engines. *Lecture Notes in Computer Science* 2527, Springer, 2002, 243-253

[11] Sicilia, M.A., García, E.: On Fuzziness in Relationship Value Segmentation: Applications to Personalized e-Commerce. *ACM SIGECOM Newsletter*, 4(2), 2003, 1–10

[12] Sicilia, M.A., García, E., Gutiérrez, J.A.: Introducing Fuzziness in Existing Orthogonal Persistence Interfaces and Systems. In: *Advances in Fuzzy Object-Oriented Databases: Modeling and Applications*, IDEA Group Publishing, 2004, 241–268

[13] Smets, P.: Imperfect information: Imprecision-Uncertainty. In: *Uncertainty Management in Information Systems: From Needs to Solutions.* Kluwer Academic Publishers, 1997, 225–254

[14] Sutton Jr., S.M. and Rouvellou, I.: Modeling Software Concerns in Cosmos. In *Proceedings of the First International Conference on Aspect–Oriented Software Development* (AOSD 2002), ACM Press, 127-133

[15] Sutton Jr, S.M. and Rouvellou, I.: Applicability of Categorization Theory to Multidimensional Separation of Concerns. In: *Proceedings of the Workshop on Advanced Separation of Concerns*, OOPSLA 2001

[16] Sutton Jr., S. M. and Tarr, P.: Aspect-Oriented Design Needs Concern Modeling. In: *Proc. of the Aspect Oriented Design Workshop on Identifying, Separating and Verifying Concerns in the Design*, Enschede, The Netherlands, 2002

[17] Yazici, A., George, R., Aksoy, D. (1998). Design and Implementation Issues in the Fuzzy Object-Oriented Data Model. *Information Sciences*, 108(1-4), 241–260

[18] Zadrozny, S., Kacprzyk, J: FQUERY for Access: Towards Human Consistent Querying User Interfaces. In: *Proceedings of the 1996 ACM Symposium on Applied Computing (SAC'96)*, 1996, 532–536